Données structurées

Christophe Alias

Objectifs

- On veut compiler des programmes avec des types structurés (tableaux, structures)
- On veut des données statiques et dynamiques

Plan

- Langage d’entrée
- Contrôle de type
- Compilation

Langage d'entrée

- P ::= TD … TD F … F
- TD ::= typedef T id;
- T ::= void | char | int | float | bool
  | T[cst] | struct { T id; … T id; }

```
typedef struct { int re; int im; } complex_t;


void add(complex_t z1, complex_t z2) {
  complex_t result = …
}
```

Langage d'entrée

- P ::= TD … TD F … F
- TD ::= typedef T id;
- T ::= void | char | int | float | bool
  | T[cst] | struct { T id; … T id; }

- F ::= T id(T id, …, T id) { D S … S }
- D ::= T id; … T id;
- E +::= E[E] | E.id | *E
- S +::= E = E

```
typedef struct { int re; int im; } complex_t;

void add(complex_t z1, complex_t z2) {
  complex_t result = …
}
```

Langage d'entrée

- P ::= TD … TD F … F
- TD ::= typedef T id;
- T ::= void | char | int | float | bool
  | T[cst] | struct { T id; … T id; }
  | *T | id Types récursifs
- F ::= T id(T id, …, T id) { D S … S }
- D ::= T id; … T id;
- E +::= E[E] | E.id | *E
- S +::= E = E

```
typedef struct { int re; int im; } complex_t;
typedef struct { int[2] item; list_t* next;} list_t;

void add(complex_t z1, complex_t z2) {
  complex_t result = …
}
```

Plan

- Langage d’entrée
- Contrôle de type
- Compilation

Objectifs

- La syntaxe donne “trop” de liberté au programmeur
- Il faut vérifier que les arguments passés aux fonctions f(E1, …, En) sont du bon type
- Il faut détecter les conversions à réaliser

```
int x;
x[0] = 1;
```

```
int char_to_int(char c)
{ return c – ‘0’; }
```

Sous-types

- T est un sous type de T' ($T \subseteq T'$) si chaque donnée T peut être convertie en donnée T'
  - char est un sous type de int
  - int est un sous type de float
- La conversion (cast) est nécessaire quand les représentations sont différentes
  - 10 (int, C2 64 bits) --> 10 (float, IEEE 754-DP)
- Un type est sous-type de lui même

Contrôle de type

- On doit vérifier que les arguments passés aux fonctions f(E1, …, En) sont du bon type.
- Il faut évaluer le type Ta de chaque argument et comparer avec le type attendu T.

Politiques:

- Typage fort: même type: Ta = T
- Typage faible: sous-type: Ta ⊆ T

Quizz

Contrôler les types avec:

- typage fort
- typage faible

```
int f(int x, int y) { … }

int main() {
  return f(‘a’,2);
}
```

Quizz, 2

```
int first(int[3] tab) {
  int r = tab[0];
  return r;
}
```

Quel est le type des fonctions = et [] ?

Types polymorphes

L'index de tableau [] est de type

[] : ∀a: a[.] x int --> a

- C'est un type polymorphe
- a est une variable de type

Contrôle des fonctions polymorphes

- On instancie les variables de type avec le contexte
- On en déduit le type retourné

```
int[3] tab;
int r = tab[0];  [] : ∀a: a[.] x int --> a
```

Unification

```
int[3] tab;
int r = tab[0];
```

[] : ∀a: a[.] x int --> a

- On cherche a et X tels que:
  int[.] x int --> X **=?** a[.] x int --> a
- S'il n'y a pas de solution, le contrôle échoue
- Sinon, X donne le type retourné
- C'est un problème d'unification de termes

Quizz

Contrôler les types pour =

```
int r = tab[0];
```

Plan

- Langage d’entrée
- Contrôle de type
- Compilation

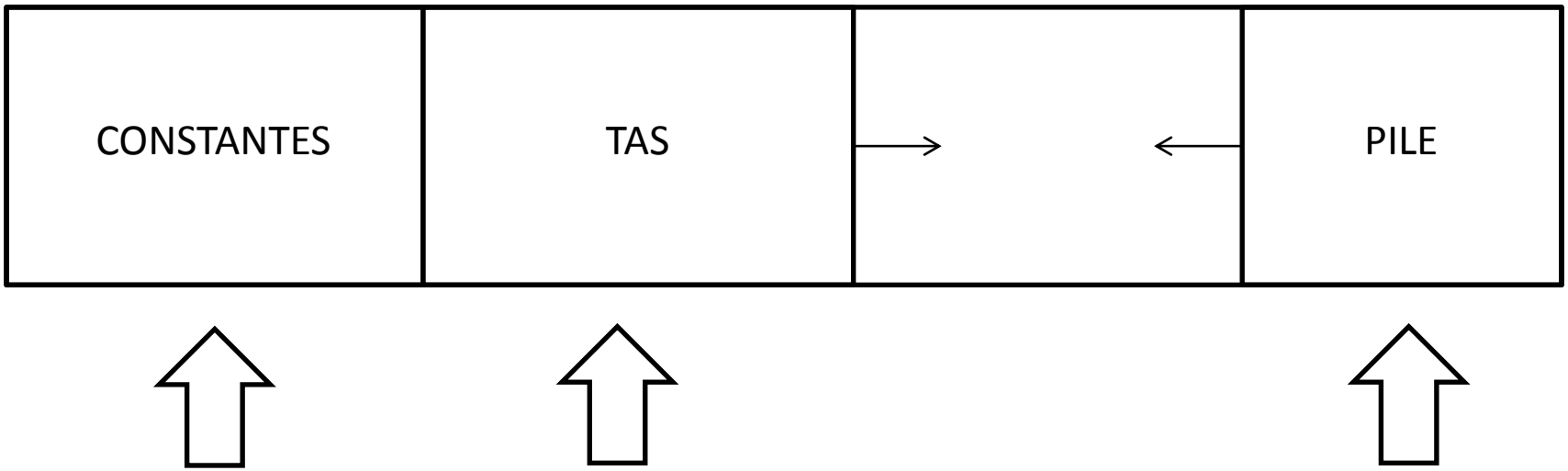
Méthodologie

Il faut spécifier:

- Où placer les données structurées (pile? tas?)
- Comment organiser une donnée (*data layout*)
- Comment construire une donnée
- Comment accéder à une donnée

```
int f(int[2] tab)
{ return tab[1]+1;
}

void main()
{ int[2] tab;
  int x;
  x = f(tab);
}
```

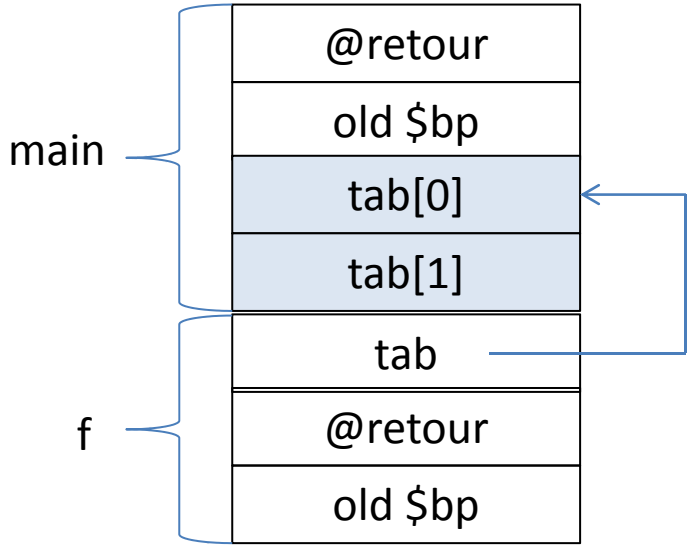
Placement des données



Extension des activations

- Les variables locales structurées sont ajoutées à l'enregistrement d'activation
- Les arguments structurés sont passés par référence

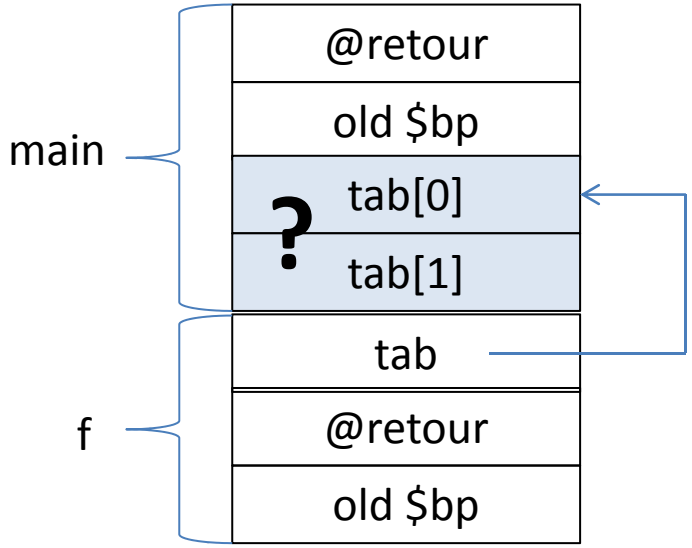
```
int f(int[2] tab)
{ return tab[1]+1;
}

void main()
{ int[2] tab;
  int x;
  x = f(tab);
}
```



Organisation des données

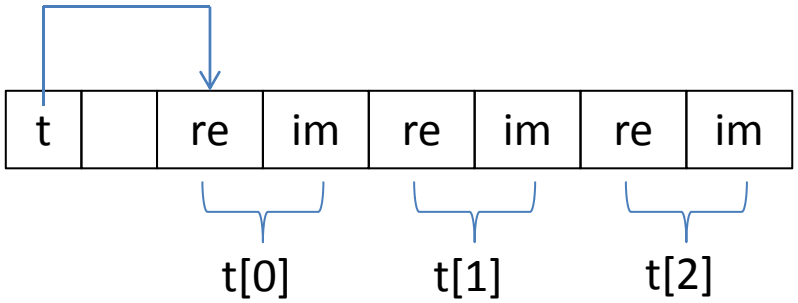
- Organisation linéaire (C)
- Organisation dirigée par la syntaxe

```
int f(int[2] tab)
{ return tab[1]+1;
}

void main()
{ int[2] tab;
  int x;
  x = f(tab);
}
```

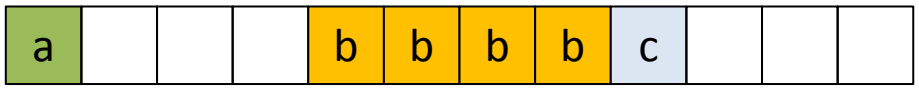


Organisation linéaire (C)

- Les tableaux et les structures sont applatis:
  struct{int re; int im}[3] t;



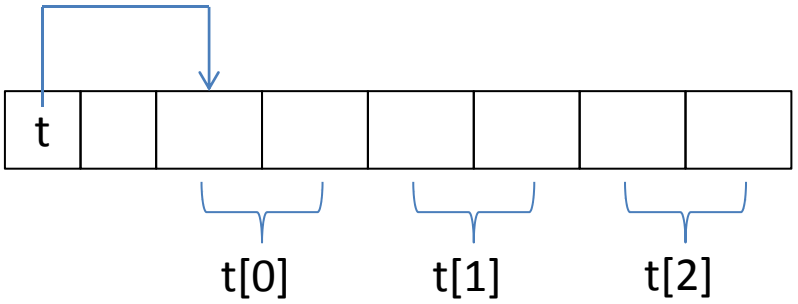
- Les structures sont alignées sur un mot:
  struct { char a; int b; char c; } s



Tableaux multidimensionnels (C)

Les tableaux sont stockés par ligne (*line-major*)

int[3][2] t;



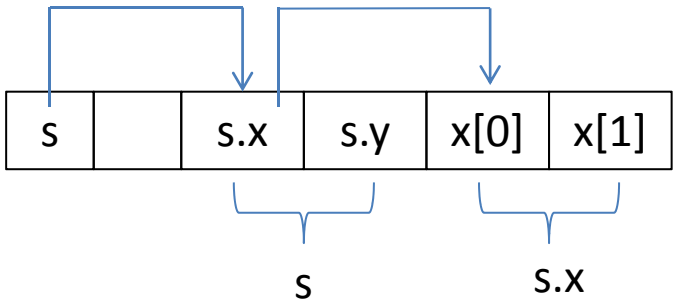
Bilan

- L'organisation linéaire ne requiert aucune construction: il suffit d'allouer l'espace.
- Elle convient donc aux données statiques et dynamiques.
- Difficulté: compiler une fonction d'accès pour chaque expression

```
struct { int im, int re} [3] t;
int sum = t[0].re + t[1].im;
```

Organisation dirigée par la syntaxe

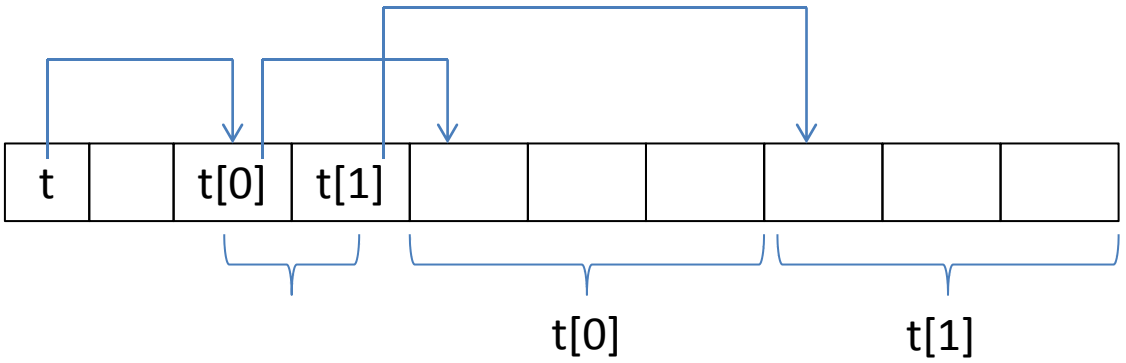
- Toute donnée structurée est masquée par une référence.
- struct{ int[2] x, int y} s
  - s est une référence vers le struct
  - s.x est une référence vers le tableau int[2] x



Organisation dirigée par la syntaxe

int[2][3] t;

- 2 tableaux de taille 3 (t[0] et t[1])
- t est une référence vers t[0]
- chaque t[i] est une référence vers un tableau int[3]



Bilan

- Les données doivent être allouées et construites (références)
- La compilation de la construction et des accés est dirigée par la syntaxe
- Il faut un accès adressé à la pile**, la machine à pile ne convient pas.**

Plan

- Langage d'entrée
- Contrôle de type
- Compilation
  - Placement des données
  - Organisation des données
  - Machine à registres
  - Compilation construction/accès

Machine à registres

- Les variables et les résultats intermédiaires sont stockés dans des registres t1, t2, …
- Pour distinguer des “vrais” registres x86, on parle de temporaires ou de registres virtuels.
- On garde la pile et les instructions
  alloc n, free, call, ret n

Entrées/Sorties

[t1+imm] = t2

Ecrit t2 à l'adresse t1+imm

t1 = [t2+imm]

Ecrit dans t1 la donnée à l'adresse t2+imm

Les adresses sont numérotées par mot

Opérations arithmétiques

t1 = imm

place imm dans t1

t1 = t2

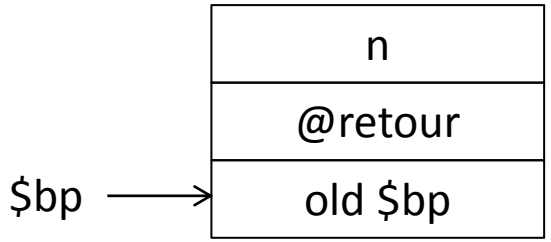
copie t2 dans t1

t1 = t2 op t3

Quizz

Que fait le programme suivant?

```
t1 = [$bp+2]
t2 = 1
t3 = t1 + t2
```



Tests

- t1 = t2 rel t3

  rel = ==, !=, <,>,…

  t1 = 1 si t1 rel t3

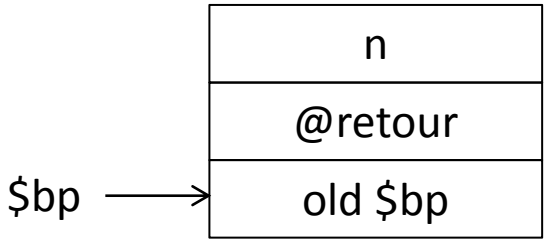
  t1 = 0 sinon

Contrôle (if, while)

jump @
  saute à @

cjump t == 0 --> @
  saute à @ si t == 0

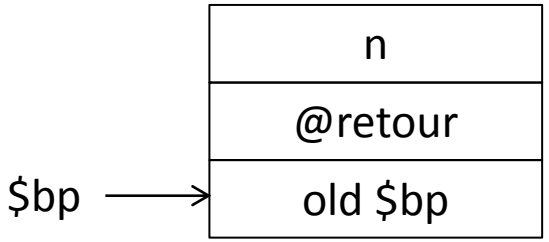
cjump t1 != 0 --> @
cjump t1 rel t2 --> @
  rel = ==,!=,<,>, …



Quizz

Que fait le code suivant?

```
t1 = [$bp+2]
cjump t1 == 0 --> then
jump else
then:
t3 = 1
else:
```



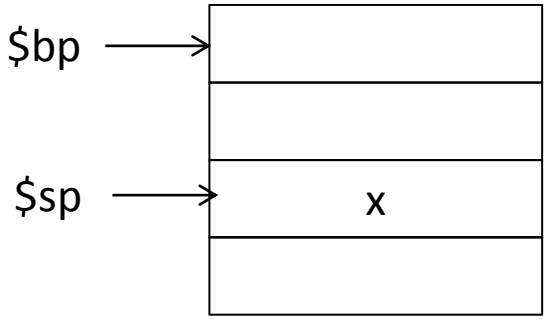
Pile

push t1

empiler(t1)

pop t1

dépile((t1)



Traduction pour la machine à registres

- $[\![E]\!]_t$ évalue E et place le résultat dans le temporaire t
- On suppose donné un environnement

  ρ : variable --> temporaire
- Les règles de traduction sont analogues à celles pour la machine à pile

Traduction des expressions

- $[\![imm]\!]_t$ =
  t = imm
- $[\![id]\!]_t$ =
  t = ρ(id)
- $[\![E1 + E2]\!]_t$ =
  $[\![E1]\!]_{t'}$
  $[\![E2]\!]_{t''}$
  t = t' + t''

Traduction des conditions

- $[\![E1>E2]\!]_t$ =
  $[\![E1]\!]_{t'}$
  $[\![E2]\!]_{t''}$
  $t = t' > t''$

Traduction du contrôle

- $[\![\text{id} = \text{E}]\!]$ =
  $[\![\text{E}]\!]_t$
  $\rho(\text{id}) = t$
- $[\![\text{if(C) S1 else S2}]\!]$ =
  $[\![\text{C}]\!]_t$
  cjump t == 0 --> else
  $[\![\text{S1}]\!]$
  jump endif
  else:
  $[\![\text{S2}]\!]$
  endif:

Quizz

Ecrire la règle de traduction du while

Traduction d'un appel de fonction

$[\![id(E1, \ldots, En)]\!]_t =$

$[\![E1]\!]_{t1}$

push t1

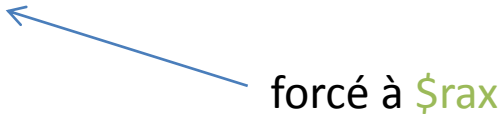
...

$[\![En]\!]_{tn}$

push tn

call id

t = t'



Traduction du return

$[\![\text{return E;}]\!]$ =

$[\![E]\!]_t$

t' = t

free

ret nb_parameters

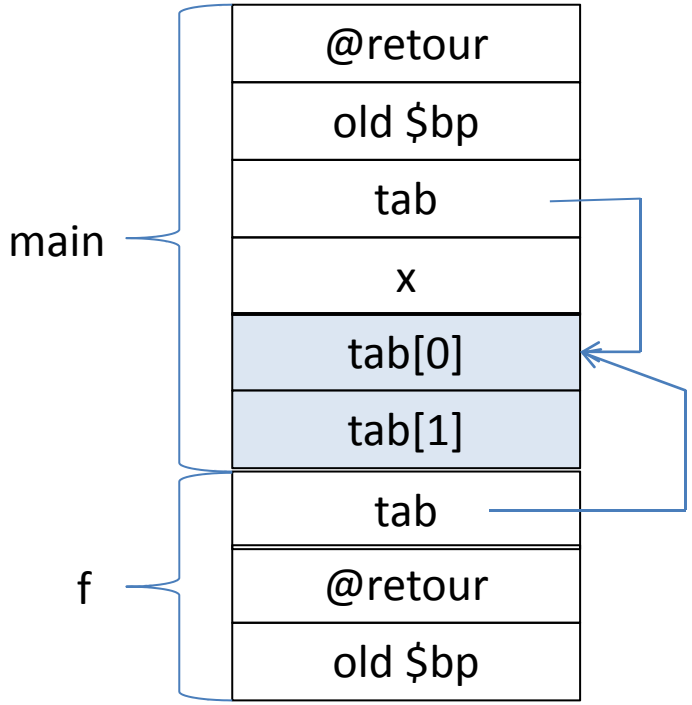
Plan

- Langage d'entrée
- Contrôle de type
- Compilation
  - Placement des données
  - Organisation des données
  - Machine à registres
  - Compilation construction/accès

Construction des données

- $[\![T]\!]_t$ ajoute une donnée de type T sur la pile et fait pointer t vers le premier élément.
- La construction est dirigée par la syntaxe

```
int f(int[2] tab)
{ return tab[1]+1;
}

void main()
{ int[2] tab;
  int x;
  x = f(tab);
}
```



Construction des structures

$[\![\text{struct } \{ \text{ T1 id1; } \dots \text{ Tn idn; } \}]\!]_t =$

t = $sp – 1

$sp = $sp – n

$[\![\text{T1}]\!]_{t1}$

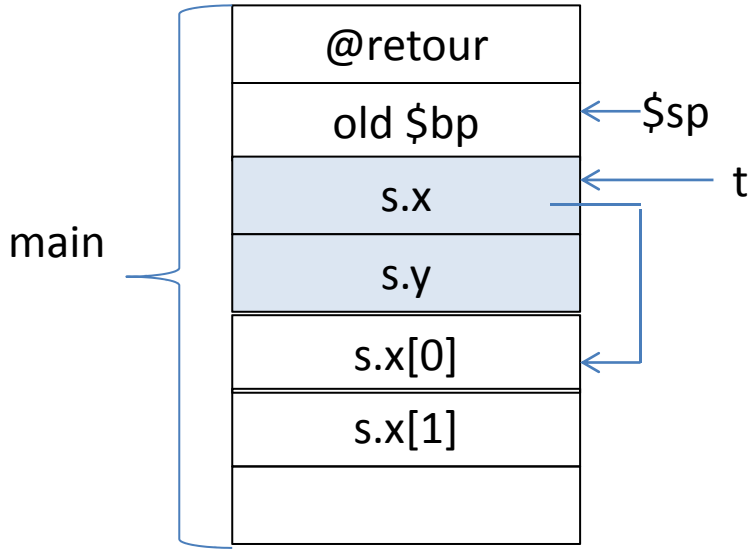
[t] = t1

…

$[\![\text{Tn}]\!]_{tn}$

[t-n+1] = tn

```
void main()
{ struct{int[2] x; int y;} s;
  …
}
```



Construction des tableaux

$[\![T[N1]...[Np]]\!]_t$ =

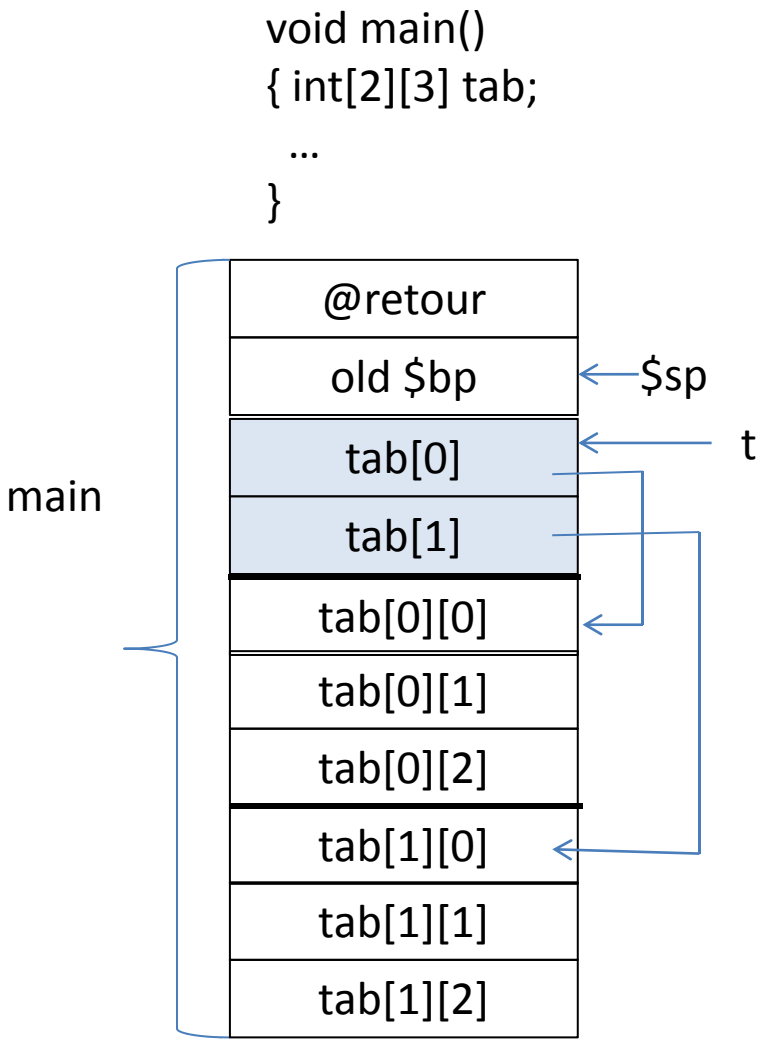
t = $sp – 1

$sp = $sp – N1

for i=1,N1:

$[\![T [N2] ... [Np]]\!]_{t'}$

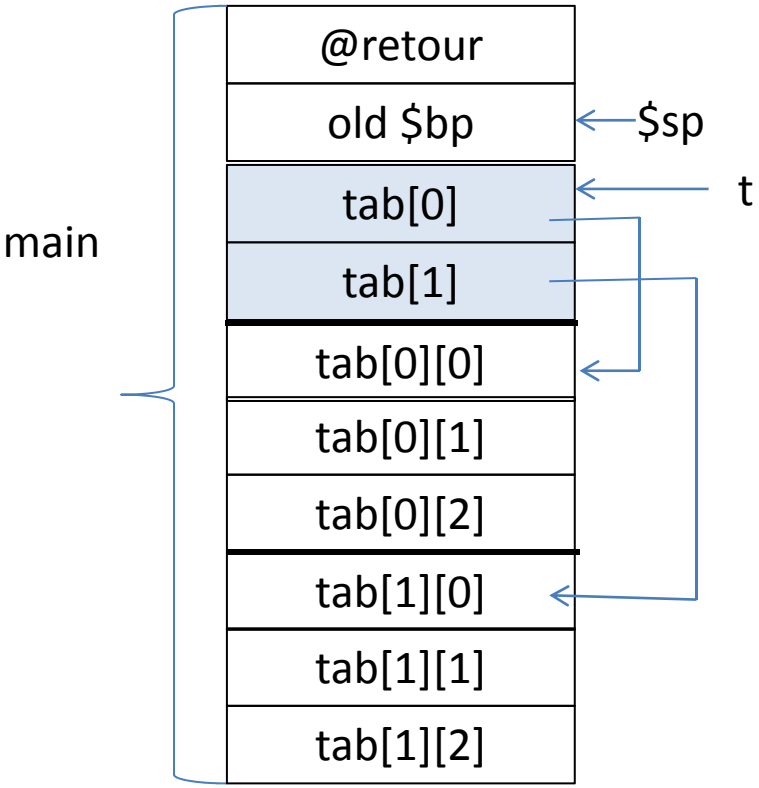
[t-i+1] = t'



Construction des tableaux

$[\![T[N1]...[Np]]\!]_t$ =
t = $sp – 1
$sp = $sp – N1
tc = t
tmin = $sp
loop:
cjump tc < tmin --> end
$[\![T [N2] ... [Np]]\!]_{t'}$
[tc] = t'
tc = tc – 1
jump loop
end:

```
void main()
{ int[2][3] tab;
  …
}
```

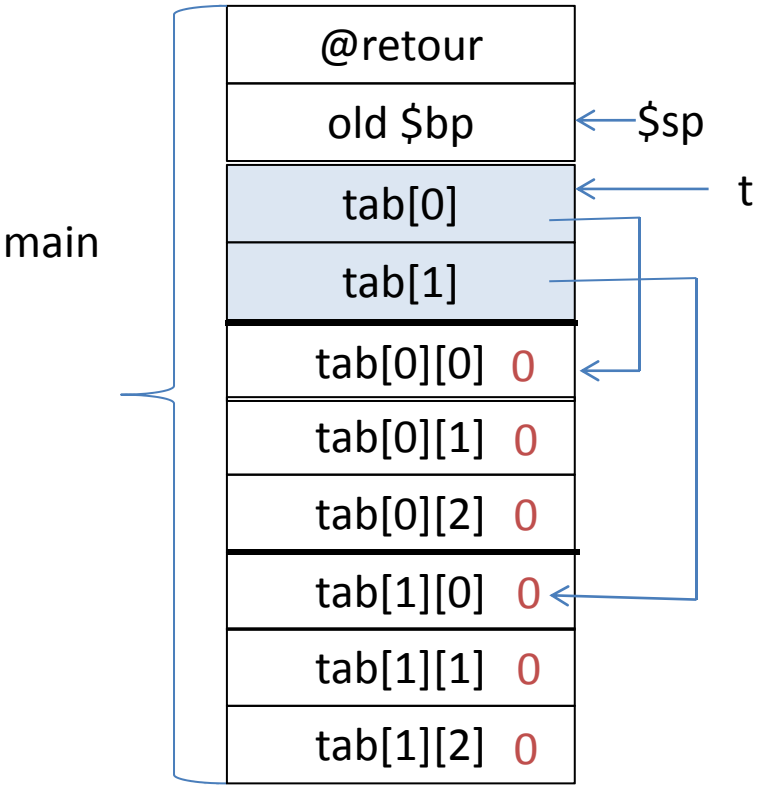


Construction des scalaires

T ::= char | int | float | bool

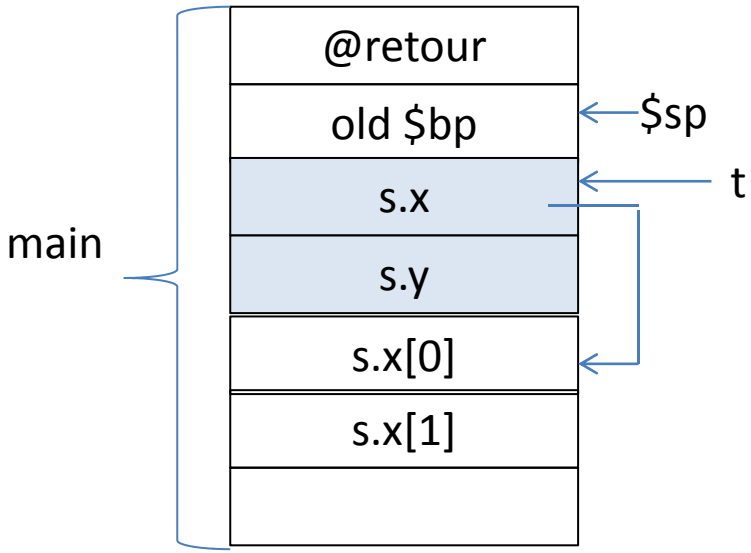
- $[\![T]\!]_t$ =
  t = 0
- $[\![*T]\!]_t$ =
  t = 0

```
void main()
{ int[2][3] tab;
  …
}
```



Quizz

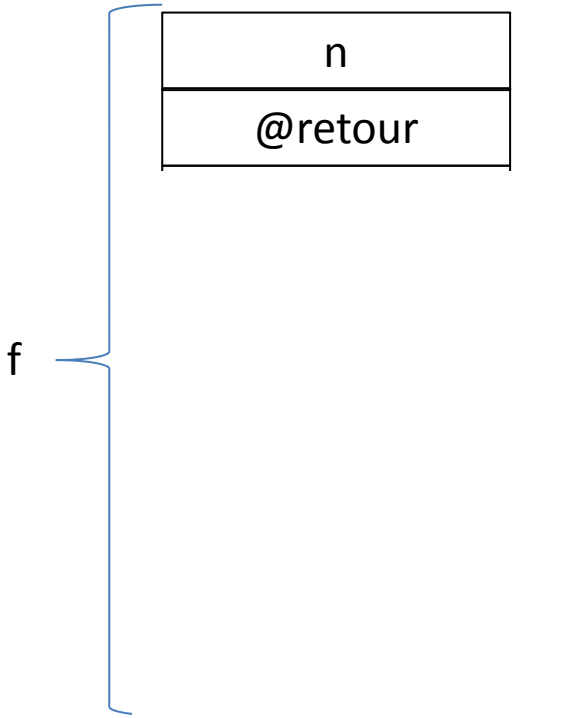
- Donner $[\![\text{struct\{int[2] x; int y;\}}]\!]_t$
- Dérouler l'exécution



Traduction des fonctions

⟦T id(T1 id1, …, tn idp) {
T’1 id’1, …, T’q id’q;
S1 … Sr }⟧ =
id:

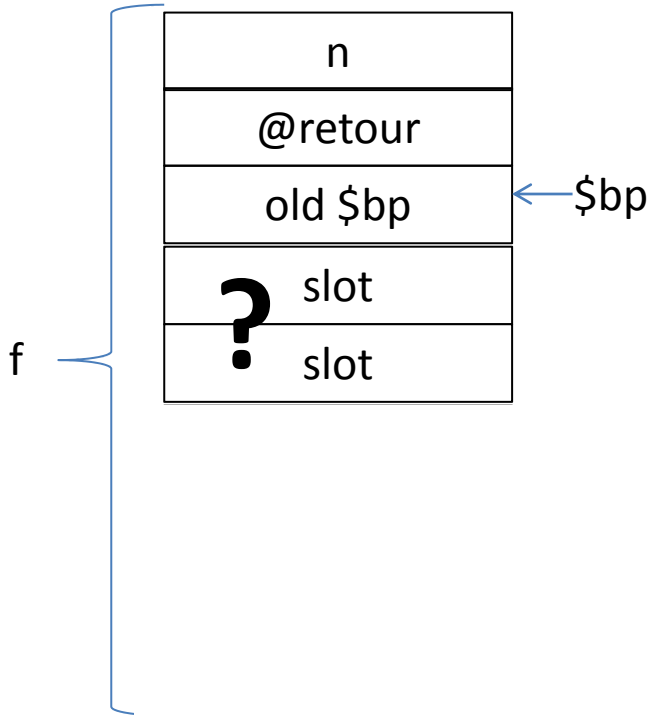
```
int f(int n)
{ struct{ int[2] x; int y; } s;
  …
}
```



Traduction des fonctions

⟦T id(T1 id1, …, tn idp) {
T’1 id’1, …, T’q id’q;
S1 … Sr }⟧ =
id:
alloc ?

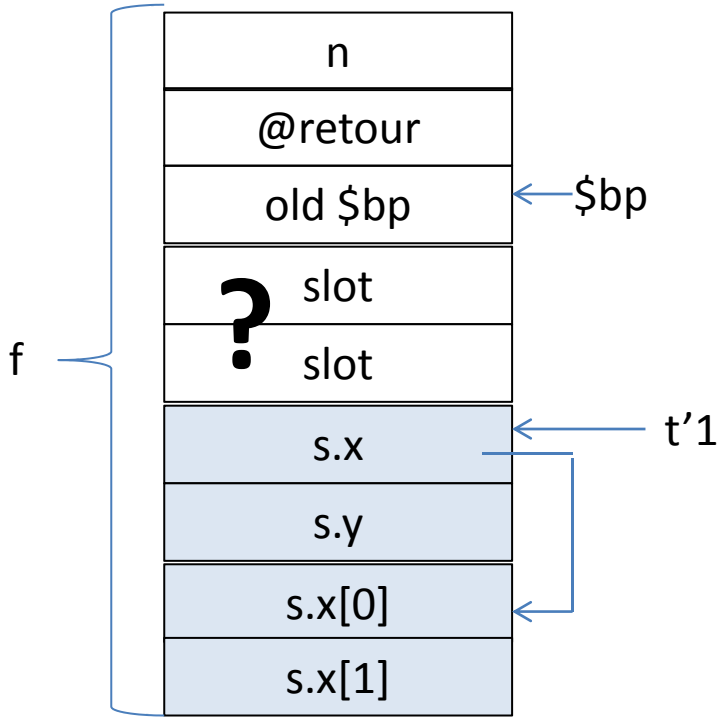
```
int f(int n)
{ struct{ int[2] x; int y; } s;
  …
}
```



Traduction des fonctions

⟦T id(T1 id1, …, tn idp) {
T’1 id’1, …, T’q id’q;
S1 … Sr }⟧ =
id:
alloc ?
$\llbracket T'1 \rrbracket_{t'1} \ldots \llbracket T'q \rrbracket_{t'q}$

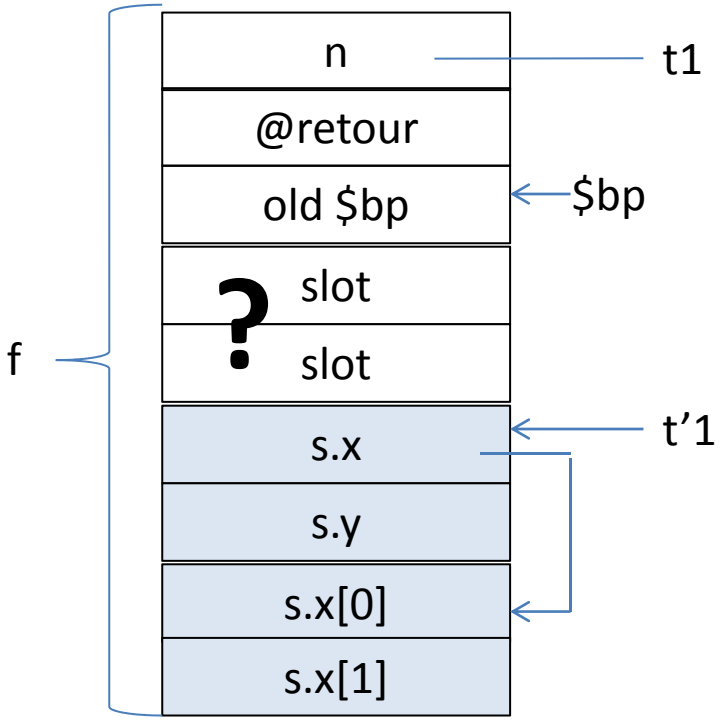
```
int f(int n)
{ struct{ int[2] x; int y; } s;
  …
}
```



Traduction des fonctions

⟦T id(T1 id1, …, tn idp) {
T’1 id’1, …, T’q id’q;
S1 … Sr }⟧ =
id:
alloc ?
$⟦T’1⟧_{t’1}$ … $⟦T’q⟧_{t’q}$
t1 = [$bp+2+p-1]
…
tp = [$bp+2]

```
int f(int n)
{ struct{ int[2] x; int y; } s;
  …
}
```



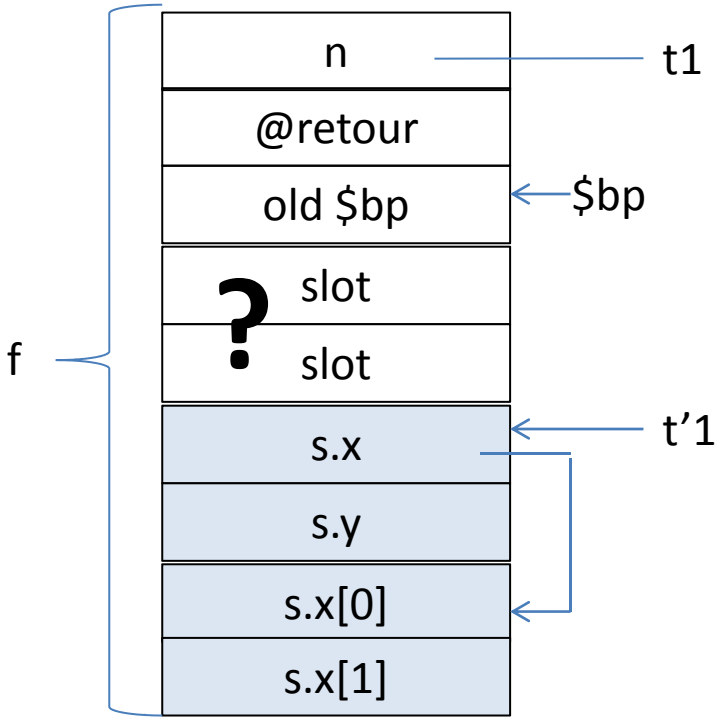
Traduction des fonctions

$[\![$T id(T1 id1, …, tn idp) {
T'1 id'1, …, T'q id'q;
S1 … Sr }$]\!]$ =

id:
alloc ?
$[\![T'1]\!]_{t'1}$ … $[\![T'q]\!]_{t'q}$
t1 = [$bp+2+p-1]
…
tp = [$bp+2]
//ρ: id1 --> t1 … idp --> tp
id'1 --> t'1 … id'q --> t'q

ρ est défini ici!

```
int f(int n)
{ struct{ int[2] x; int y; } s;
  …
}
```



Traduction des fonctions

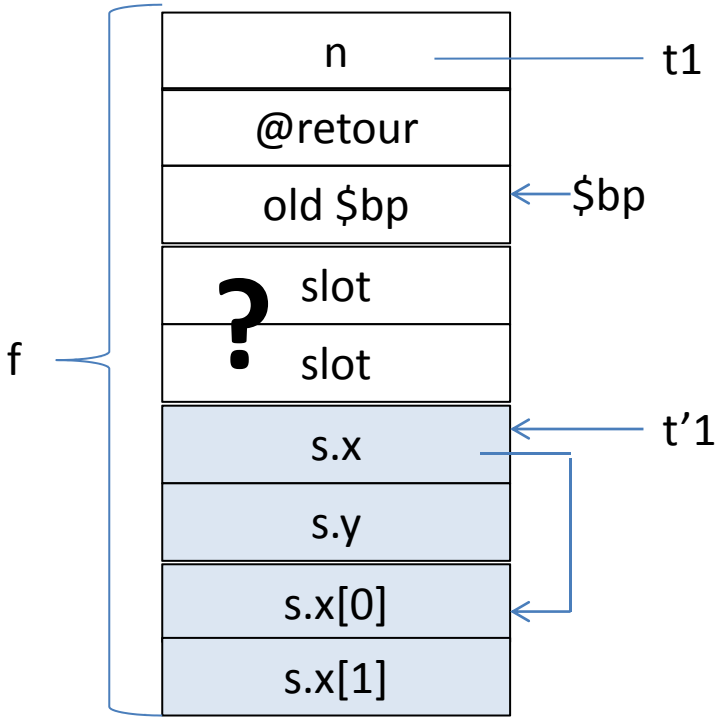
⟦T id(T1 id1, …, tn idp) {
T’1 id’1, …, T’q id’q;
S1 … Sr }⟧ =

id:
alloc ?
$⟦T'1⟧_{t'1} \dots ⟦T'q⟧_{t'q}$
t1 = [$bp+2+p-1]
…
tp = [$bp+2]
//ρ: id1 --> t1 … idp --> tp
id’1 --> t’1 … id’q --> t’q

⟦S1⟧ … ⟦Sr⟧

free
ret p

```
int f(int n)
{ struct{ int[2] x; int y; } s;
  …
}
```



Exemple

f:

alloc ?

t1 = [$bp+2]

$[\![\text{struct}\{\text{int}[2]\ x;\ \text{int}\ y;\ \}]\!]_{t'1}$

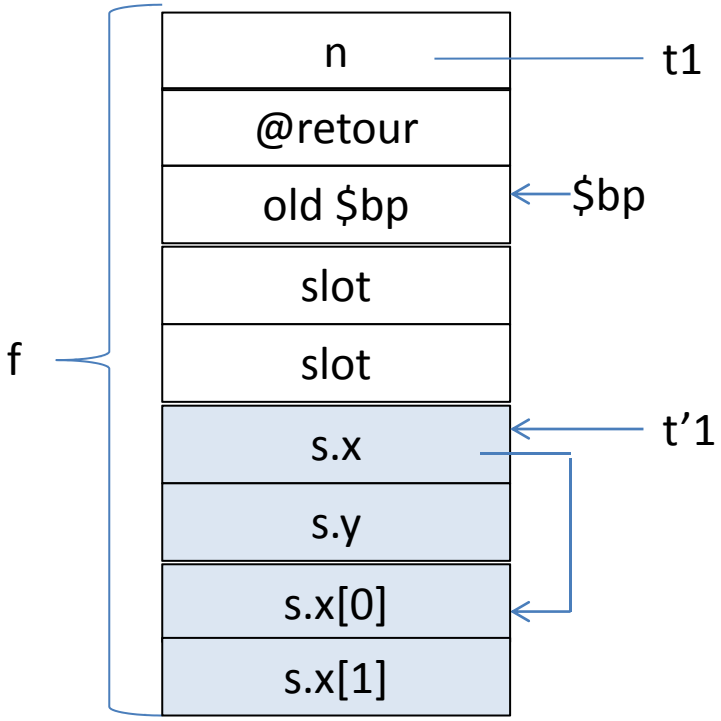
//ρ: n --> t1, s --> t'1

$[\![S1]\!]$ … $[\![Sr]\!]$

free

ret p

```
int f(int n)
{ struct{ int[2] x; int y; } s;
  …
}
```



Exemple

f:

alloc ?

t1 = [$bp+2]

$[\![\text{struct\{int[2] x; int y; }\}\!]_{t'1}$

- t'1 = $sp – 1
- $sp = $sp – 2
- $[\![\text{int[2]}]\!]_{t'2}$
- [t'1] = t'2
- $[\![\text{int}]\!]_{t'3}$
- [t'1-1] = t'3

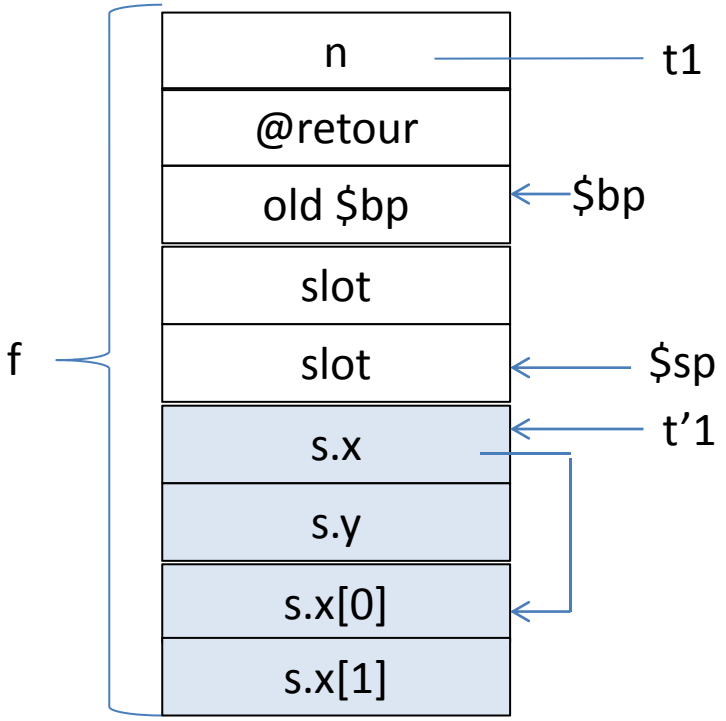
//ρ: n --> t1, s --> t'1

$[\![S1]\!]$ … $[\![Sr]\!]$

free

ret p

```
int f(int n)
{ struct{ int[2] x; int y; } s;
  …
}
```



Exemple

f:

alloc ?

t1 = [$bp+2]

$\llbracket$struct{int[2] x; int y; }$\rrbracket_{t'1}$

- t'1 = $sp – 1
- $sp = $sp – 2
- $\llbracket$int[2]$\rrbracket_{t'2}$
  - t'2 = $sp – 1
  - $sp = $sp – 2
- [t'1] = t'2
- $\llbracket$int $\rrbracket_{t'3}$
  - t'3 = 0
- [t'1-1] = t'3

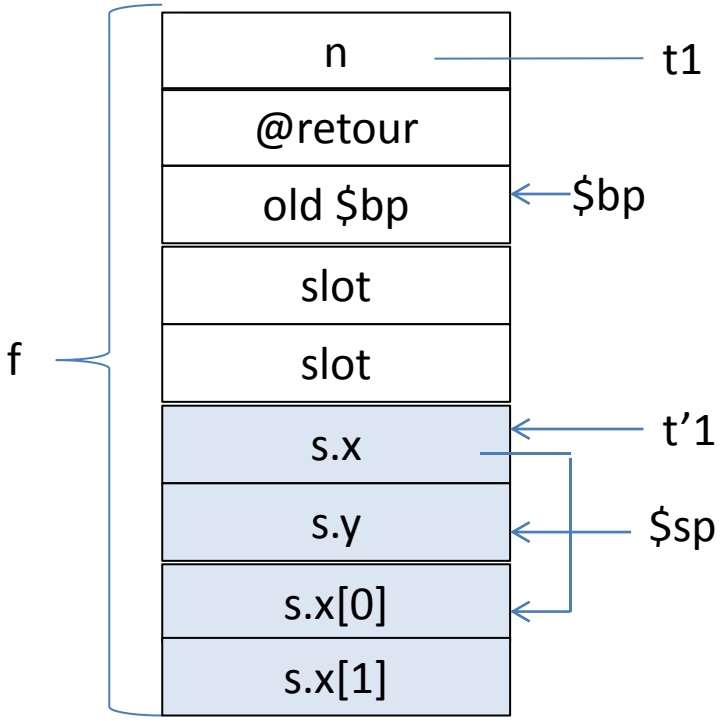
//ρ: n --> t1, s --> t'1

$\llbracket$S1$\rrbracket$ … $\llbracket$Sr$\rrbracket$

free

ret p

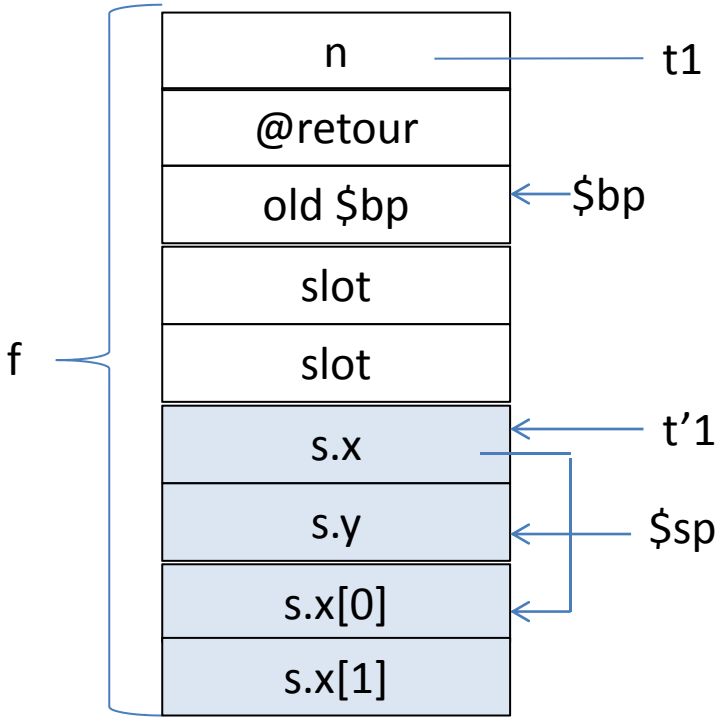
```
int f(int n)
{ struct{ int[2] x; int y; } s;
  …
}
```



Exemple

```
f:
  alloc ?
  t1 = [$bp+2]
  t'1 = $sp – 1
  $sp  = $sp – 2
  t'2 = $sp – 1
  $sp = $sp – 2
  [t'1] = t'2
  t'3 = 0
  [t'1-1] = t'3
  //ρ: n --> t1, s --> t'1

  ⟦S1⟧ ... ⟦Sr⟧

  free
  ret p
```

```
int f(int n)
{ struct{ int[2] x; int y; } s;
  …
}
```



Traduction des accés

Pour traduire a[i+1] = 2*b[i] + 1

Il faut calculer:

- l'adresse de a[i+1] (*L-value*)
- la valeur de 2*b[i] + 1 (*R-value*)

La traduction d'un accés doit calculer ces valeurs:

$[\![E]\!]_{ta,tv}$

Traduction des affectations (structurées)

$[\![E1 = E2]\!] =$

$[\![E1]\!]_{ta,tv}$

$[\![E2]\!]_{t}$

$[ta] = t$

Traduction d'un accès tableau

$[\![E1[E2]]\!]_{ta,tv}$ =

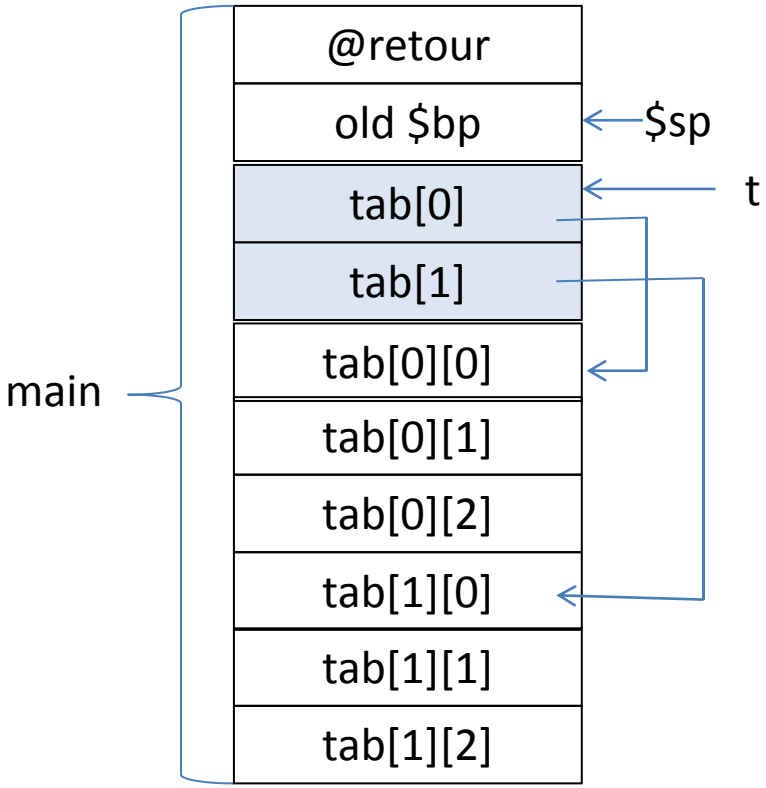
$[\![E1]\!]_{tbase}$

$[\![E2]\!]_{toffset}$

ta = tbase - toffset

tv = [ta]

```
void main()
{ int[2][3] tab;
  …
}
```



Traduction d'un accès tableau

$[\![E1[E2]]\!]_{ta,tv}$ =

$[\![E1]\!]_{tbase}$

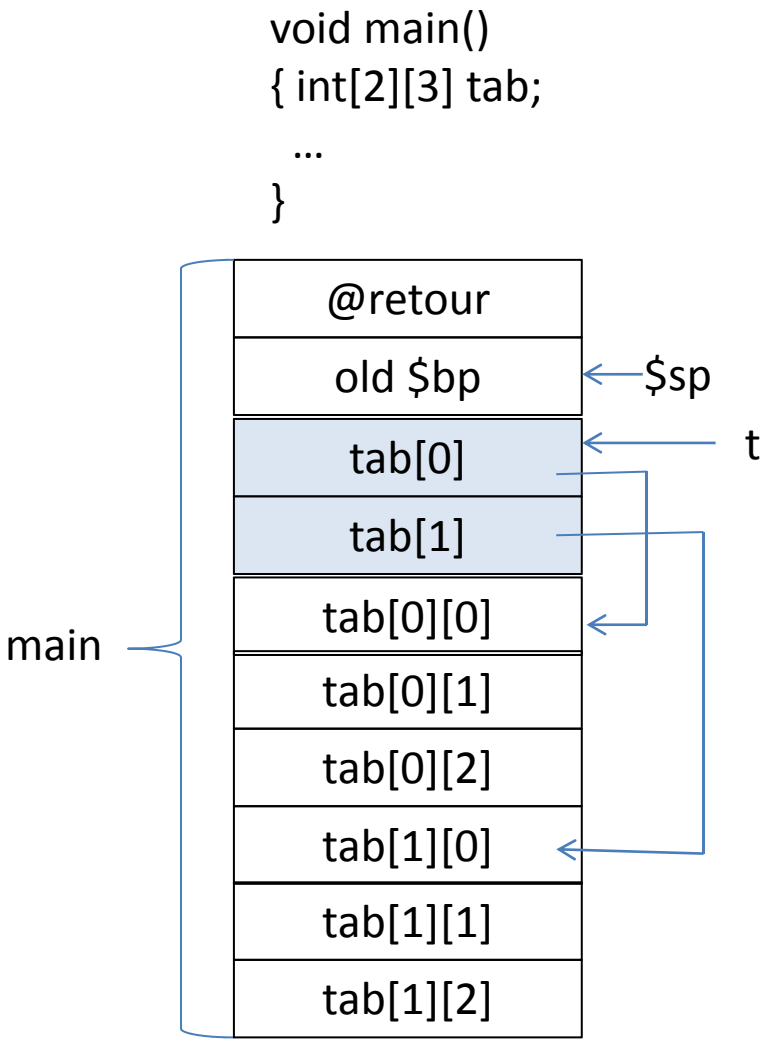
$[\![E2]\!]_{toffset}$

ta = tbase - toffset

tv = [ta]

Si E est un accès,

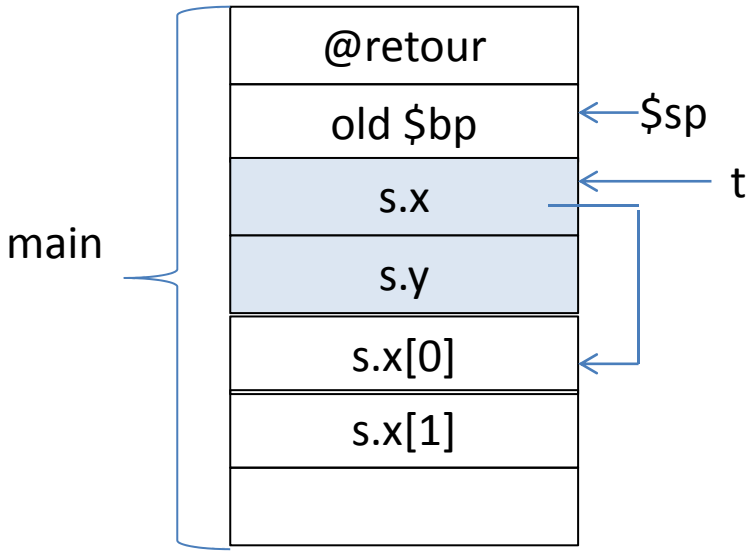
- $[\![E]\!]_t = [\![E]\!]_{ta,t}$
- On ignore ta



Traduction d'un accès struct

$[\![E.id]\!]_{ta,tv} =$
$[\![E]\!]_{tbase}$
toffset = rang(id)
ta = tbase - toffset
tv = [ta]

```
void main()
{ struct{int[2] x; int y;} s;
  …
}
```



Traduction d’un accès pointeur

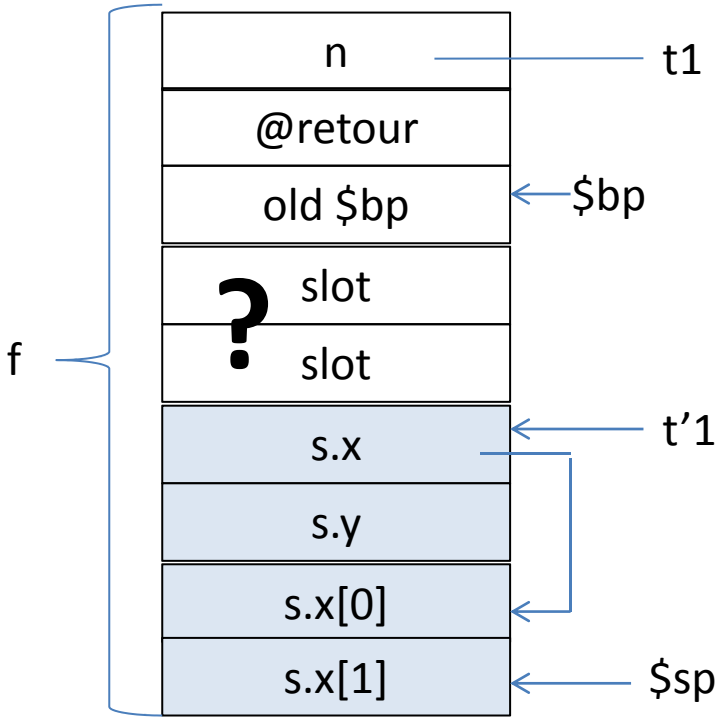
$[\![*E]\!]_{ta,tv} =$
$[\![E]\!]_{ta}$
$tv = [ta]$

That’s it!

Exemple, fin

```
f:
  alloc ?
  t1 = [$bp+2]
  t'1 = $sp – 1
  $sp  = $sp – 2
  t'2 = $sp – 1
  $sp = $sp – 2
  [t'1] = t'2
  t'3 = 0
  [t'1-1] = t'3
  //ρ: n --> t1, s --> t'1

  ⟦s.x[1] = n;⟧

  free
  ret p
```

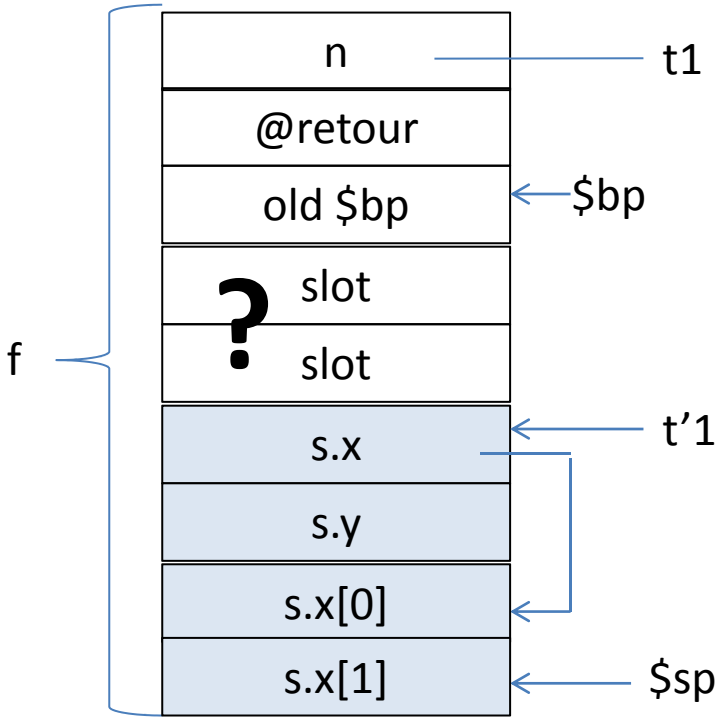
```
int f(int n)
{ struct{ int[2] x; int y; } s;
  s.x[1] = n;
}
```



Exemple, fin

```
f:
  alloc ?
  t1 = [$bp+2]
  t'1 = $sp – 1
  $sp  = $sp – 2
  t'2 = $sp – 1
  $sp = $sp – 2
  [t'1] = t'2
  [t'1-1] = t'3
  //ρ: n --> t1, s --> t'1

  ⟦(s.x)[1]⟧ta,tv
  ⟦n⟧t
  [ta] = t

  free
  ret p
```

```
int f(int n)
{ struct{ int[2] x; int y; } s;
  s.x[1] = n;
}
```



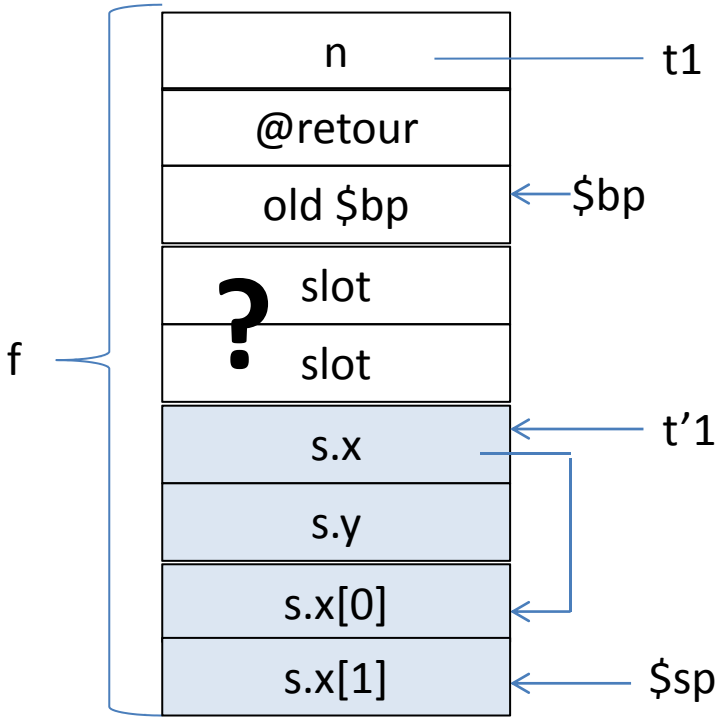
Exemple, fin

```
f:
  alloc ?
  t1 = [$bp+2]
  t'1 = $sp – 1
  $sp  = $sp – 2
  t'2 = $sp – 1
  $sp = $sp – 2
  [t'1] = t'2
  t'3 = 0
  [t'1-1] = t'3
  //ρ: n --> t1, s --> t'1

  ⟦(s.x)[1]⟧ta,tv
  ⟦n⟧t
  [ta] = t

  free
  ret p
```

$⟦s.x⟧_{tbase}$
$⟦1⟧_{tbase}$
ta = tbase - toffset
tv = [ta]

t = t1

```
int f(int n)
{ struct{ int[2] x; int y; } s;
  s.x[1] = n;
}
```



Exemple, fin

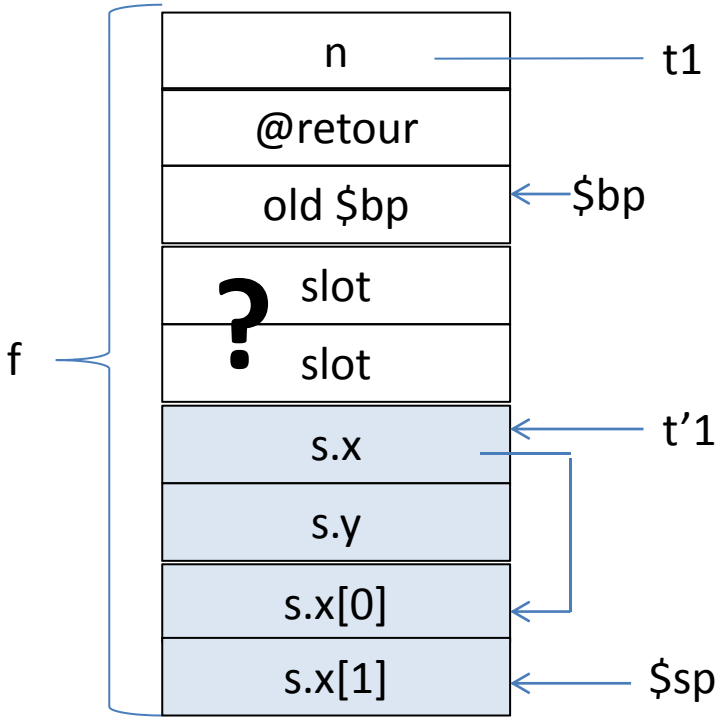
```
f:
  alloc ?
  t1 = [$bp+2]
  t'1 = $sp – 1
  $sp  = $sp – 2
  t'2 = $sp – 1
  $sp = $sp – 2
  [t'1] = t'2
  t'3 = 0
  [t'1-1] = t'3
  //ρ: n --> t1, s --> t'1

  ⟦(s.x)[1]⟧ta,tv
  ⟦n⟧t
  [ta] = t

  free
  ret p
```

On ignore ta'

$⟦s.x⟧_{ta',tbase}$
$⟦1⟧_{toffset}$
ta = tbase - toffset
tv = [ta]

t = t1

```
int f(int n)
{ struct{ int[2] x; int y; } s;
  s.x[1] = n;
}
```



Exemple, fin

```
f:
  alloc ?
  t1 = [$bp+2]
  t'1 = $sp – 1
  $sp  = $sp – 2
  t'2 = $sp – 1
  $sp = $sp – 2
  [t'1] = t'2
  t'3 = 0
  [t'1-1] = t'3
  //ρ: n --> t1, s --> t'1

  ⟦(s.x)[1]⟧ta,tv
  ⟦n⟧t
  [ta] = t

  free
  ret p
```

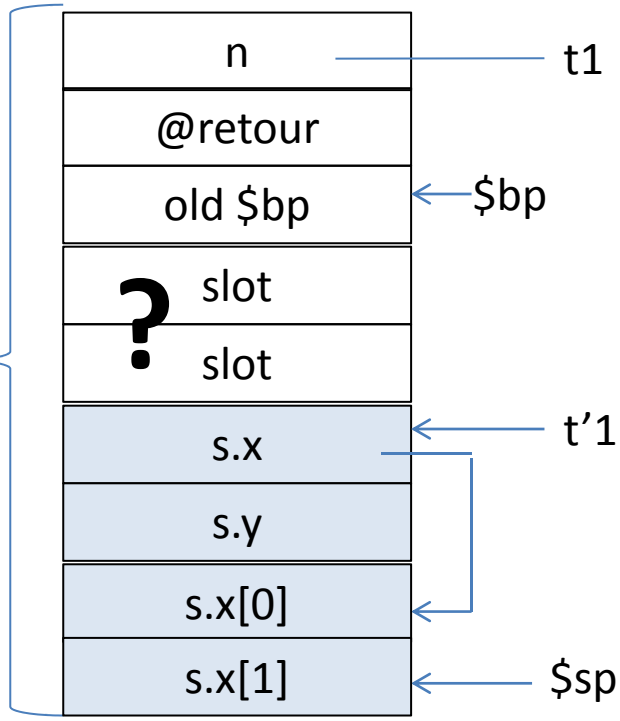
$⟦s.x⟧_{ta',tbase}$
$⟦1⟧_{toffset}$
ta = tbase - toffset
tv = [ta]

$⟦s⟧_{tbase'}$
toffset' = 0
ta' = tbase' – toffset'
tbase = [ta']

toffset = 1

t = t1

```
int f(int n)
{ struct{ int[2] x; int y; } s;
  s.x[1] = n;
}
```



Exemple, fin

```
f:
  alloc ?
  t1 = [$bp+2]
  t'1 = $sp – 1
  $sp  = $sp – 2
  t'2 = $sp – 1
  $sp = $sp – 2
  [t'1] = t'2
  t'3 = 0
  [t'1-1] = t'3
  //ρ: n --> t1, s --> t'1

  ⟦(s.x)[1]⟧ta,tv
  ⟦n⟧t
  [ta] = t

  free
  ret p
```

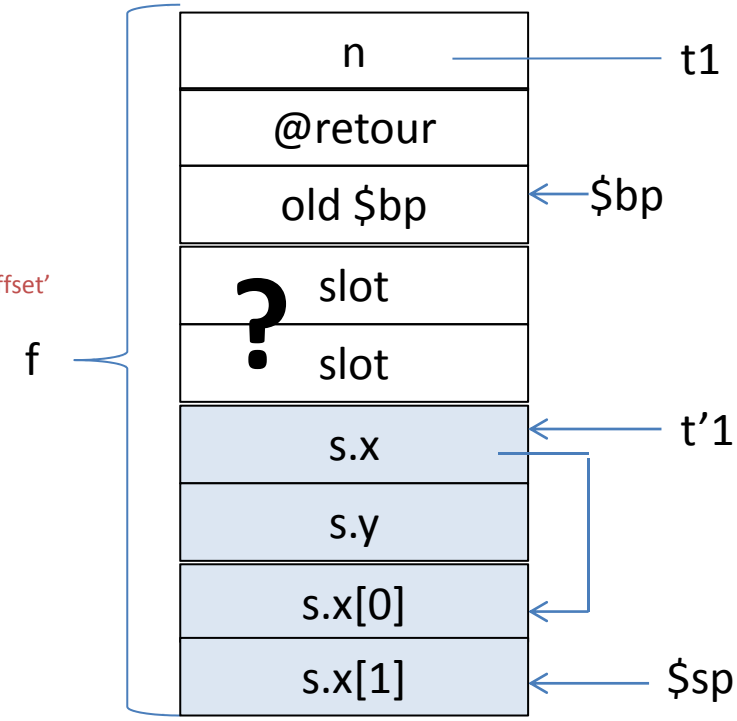
⟦s.x⟧ta',tbase
⟦1⟧toffset
ta = tbase - toffset
tv = [ta]

tbase' = t'1
toffset' = 0
ta' = tbase' – toffset'
tbase = [ta']

toffset = 1

t = t1

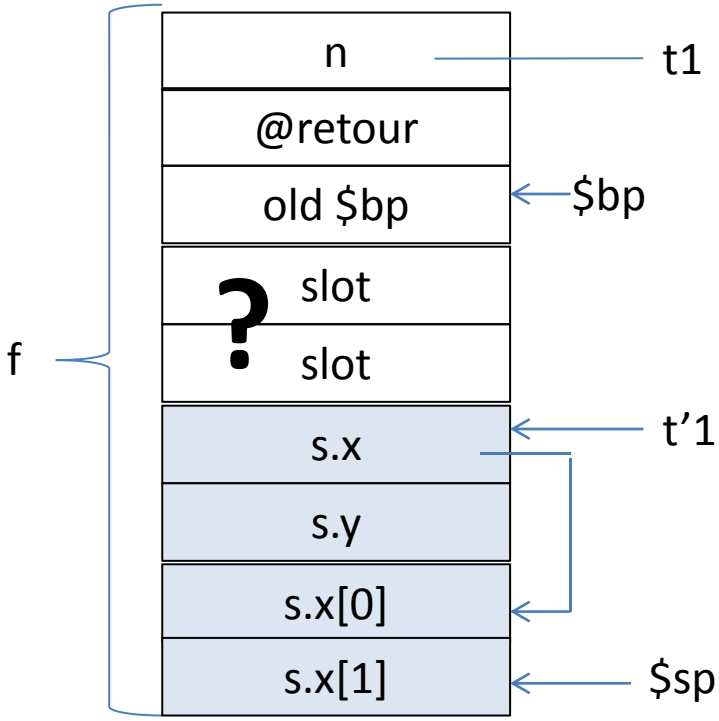
```
int f(int n)
{ struct{ int[2] x; int y; } s;
  s.x[1] = n;
}
```



Exemple, fin

```
f:
  alloc ?
  t1 = [$bp+2]
  t'1 = $sp – 1
  $sp  = $sp – 2
  t'2 = $sp – 1
  $sp = $sp – 2
  [t'1] = t'2
  t'3 = 0
  [t'1-1] = t'3
  //ρ: n --> t1, s --> t'1

  tbase' = t'1
  toffset' = 0
  ta' = tbase' – toffset'
  tbase = [ta']
  toffset = 1
  ta = tbase - toffset
  tv = [ta]
  t = t1
  [ta] = t

  free
  ret p
```

```
int f(int n)
{ struct{ int[2] x; int y; } s;
  s.x[1] = n;
}
```



Et ensuite?

```
f:
  alloc ?
  t1 = [$bp+2]
  t'1 = $sp – 1
  $sp  = $sp – 2
  t'2 = $sp – 1
  $sp = $sp – 2
  [t'1] = t'2
  t'3 = 0
  [t'1-1] = t'3
  //ρ: n --> t1, s --> t'1

  tbase' = t'1
  toffset' = 0
  ta' = tbase' – toffset'
  tbase = [ta']
  toffset = 1
  ta = tbase - toffset
  tv = [ta]
  t = t1
  [ta] = t

  free
  ret p
```